Necessary and Sufficient Preconditions via Eager Abstraction

Necessary and Sufficient Preconditions via Eager Abstraction *

Mohamed Nassim Seghir[1] and Peter Schrammel[2]

[1] University of Edinburgh
[2] University of Oxford

Abstract. The precondition for safe execution of a procedure is useful for understanding, verifying and debugging programs. We have previously presented a CEGAR-based approach for inferring necessary and sufficient preconditions based on the iterative abstraction-refinement of the set of safe and unsafe states until they become disjoint. A drawback of that approach is that safe and unsafe traces are explored separately and each time they are built entirely before being checked for consistency. In this paper, we present an *eager* approach that explores shared prefixes between safe and unsafe traces conjointly. As a result, individual state sets, by construction, fulfil the property of separation between safe and unsafe states without requiring any refinement. Experiments using our implementation of this technique in the precondition generator P-Gen show a significant improvement compared to our previous CEGAR-based method. In some cases the running time drops from several minutes to several seconds.

1 Introduction

Procedure preconditions must hold when invoking a procedure in order to guarantee its intended, safe behaviour during its execution. They are an important concept in design-by-contract, and commonly found in code documentation, e.g. for libraries, in order to help the developer understand how to use a procedure in the current calling context.

However, it is notoriously difficult to come up with preconditions that guarantee that all assertions in the procedure hold under all possible inputs that satisfy them (*sufficient preconditions*), but, at the same time, do not rule out safe behaviour (*necessary preconditions*).

Computing the weakest sufficient or strongest necessary preconditions syntactically is not always possible as programs (due to loops) often contain an infinite number of paths. On the one hand, over-approximating these infinite sets may include unsafe paths which lead to the violation of the assertion and thus giving an unsound result. On the other hand, under-approximating them

* The first author was supported by EPSRC under grant number EP/K032666/1 "App Guarden". The second author was supported by the ARTEMIS Joint Undertaking under grant number 295311 "VeTeSS".

may exclude safe paths which might rule out desirable safe behaviour and hence render the precondition unusable. *True*, the precondition that allows all traces of the program, is always a valid (over-approximating) necessary precondition, and *false*, the precondition that forbids program execution, is always a valid (under-approximating) sufficient precondition. Obviously, the former is not sound and the latter is not useful, in general.

In our previous work [29], we proposed a solution to this problem based on iteratively abstracting both the set of safe and unsafe states and refining them until they become disjoint. Thus, the resulting precondition is sufficient and also necessary for the validity of the assertions. This guarantees the absence of false alarms. Of course, this is only possible if the precondition is expressible in the chosen abstract domain (or predicate language); otherwise the algorithm fails to find a suitable precondition. A disadvantage of that approach is the exploration of safe and unsafe states separately. Hence, we do not know the frontier between safe and unsafe states to guide the abstraction at early stages and the refinement is only applied after entire traces are built. Moreover, this laziness in the abstraction process introduces redundant computation steps which can be avoided if safe and unsafe states are explored conjointly.

In this paper, we present an eager approach for inferring necessary and sufficient preconditions in a monotonic fashion[3]. Based on the observation that safe and unsafe traces share most of their prefixes and only differ by small portions in the traces, our approach explores safe and unsafe states conjointly as pairs. Hence the criterion for guiding the abstraction is that each two elements forming pairs of safe and unsafe states at a given location must be disjoint. This new procedure has many advantages:

- Inferring relevant and general predicates at early stages, hence boosting the convergence of the fix-point computation.
- By construction, states fulfil the global constraint of separating the set of safe states from unsafe ones. Hence, the refinement process is totally skipped and a series of iteration steps are avoided.
- Computational redundancies are eliminated as shared prefixes between safe and unsafe traces are explored conjointly.

The inferred preconditions have the same expressiveness as those obtained by our previous method [29], however, the new approach exhibits an enormous improvement in algorithmic efficiency.

We have implemented our approach in the precondition generator P-Gen and performed a comparative study with our CEGAR-based technique. The results clearly demonstrate that our new method is not just a side optimisation but rather represents the right way to proceed for inferring necessary and sufficient preconditions. For all the programs we have tested, the eager approach performs better than the lazy (CEGAR) one. For some cases where the lazy approach takes several minutes, the eager one just requires several seconds.

[3] By monotonic, we mean that the set of states will only increase.

The remainder of the paper is organised as follows: Section 2 illustrates the intuition behind our approach through an example. Section 3 introduces some preliminary material. Section 4 formally exposes our precondition inference approach. Section 5 presents an experimental comparative study and Section 6 surveys related work.

2 Example

To highlight the advantages of our new approach for precondition inference, we briefly recall our previous work [29] and illustrate both techniques on the procedure copy in Figure 1. The procedure takes as parameters two arrays a and b, and copies a range of elements of b to the corresponding range in array a. The access to array a is safe if the index expression is in the range $[0..a\_l - 1]$, where a_l is the length of array a. It is trivial to see that the lower bound is not violated. The safety condition for the upper bound is expressed by the assertion at location ℓ_2. Our goal is to find a necessary and sufficient precondition for procedure copy which guarantees that this assertion is never violated. It means that it should neither be too strong nor too weak. To ease the presentation, we solely focus on the specified assertion, assuming that there are no other run-time exceptions caused by null dereferences, i.e., $a \neq null$ and $b \neq null$.

```
void copy(int a[], int b[])
{
    int i;
ℓ0 : i = 0;
ℓ1 : while(b[i] != 0)
    {
ℓ2 :    assert(i < a_l);
        a[i] = b[i];
        i++;
    }
}
```

Fig. 1. A simple program that copies a range of elements from array b to array a. The limit of the range to be copied is implicitly delimited via the sentinel value 0, and a_l is the length of array a.

For illustration, we formally represent programs in terms of transition constraints over primed and unprimed program variables. The set of transition constraints corresponding to program copy (Figure 1) is given in Figure 2(a) and the associated control flow graph is given in Figure 2(b). The program counter is modelled explicitly using the variable pc, which ranges over the set of control locations. The assertion in the original program is replaced with a conditional

branch whose condition is the negation of the assertion and whose target is the *error* location ℓ_E. The special location ℓ_F is the *final* location, and has no successor. Observe that the error location is only reachable if $i \geq a\_l$ evaluates to true at location ℓ_2. The final location ℓ_F is reached in paths without error. Arrays a and b are represented by uninterpreted function symbols, and $a[x := e]$ denotes function update (the expression is equal to a where the x^{th} element has been replaced by e).

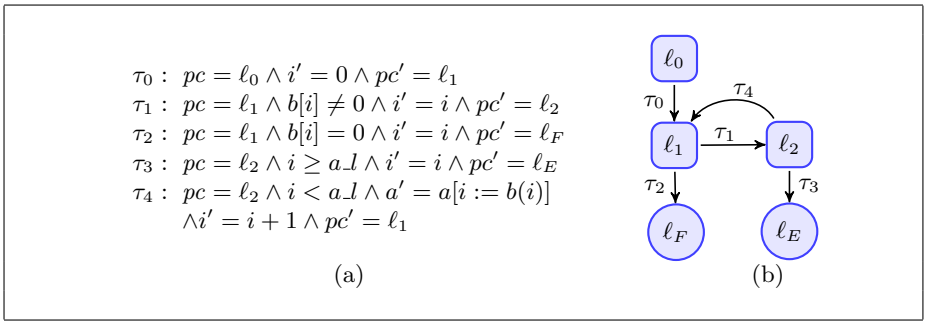

Fig. 2. Transition constraints for program copy (a) and the corresponding control flow graph (b).

CEGAR-based precondition inference. The CEGAR-based approach for precondition generation consists of building abstractions of safe and unsafe states and refining them until they become disjoint. It mainly comprises the following steps:

1. Build abstraction: abstract both the set of safe and unsafe states.
2. Find a counterexample: two abstract traces, a safe one and an unsafe one, beginning with a common initial state.
3. Check counterexample: checks if the two traces can be concretised in the original program. The check is carried out by computing the weakest precondition for each trace.
4. Refine: the spurious counterexample is ruled out by adding predicates that refine the abstraction making the two traces no longer sharing their initial state.

In the refinement phase (steps 3 and 4), safe and unsafe traces are separately explored backwards, and the consistency check is only applied when the initial location is reached. Considering the example of Figure 1, let us assume that the safe trace $\langle \tau_0, \tau_1, \tau_4, \tau_2 \rangle$ and the unsafe one $\langle \tau_0, \tau_1, \tau_4, \tau_1, \tau_3 \rangle$ are generated by entering the loop once. The backward analysis of these two traces is illustrated in Figure 3. On the left (a) is the safe trace and on the right (b) is the unsafe

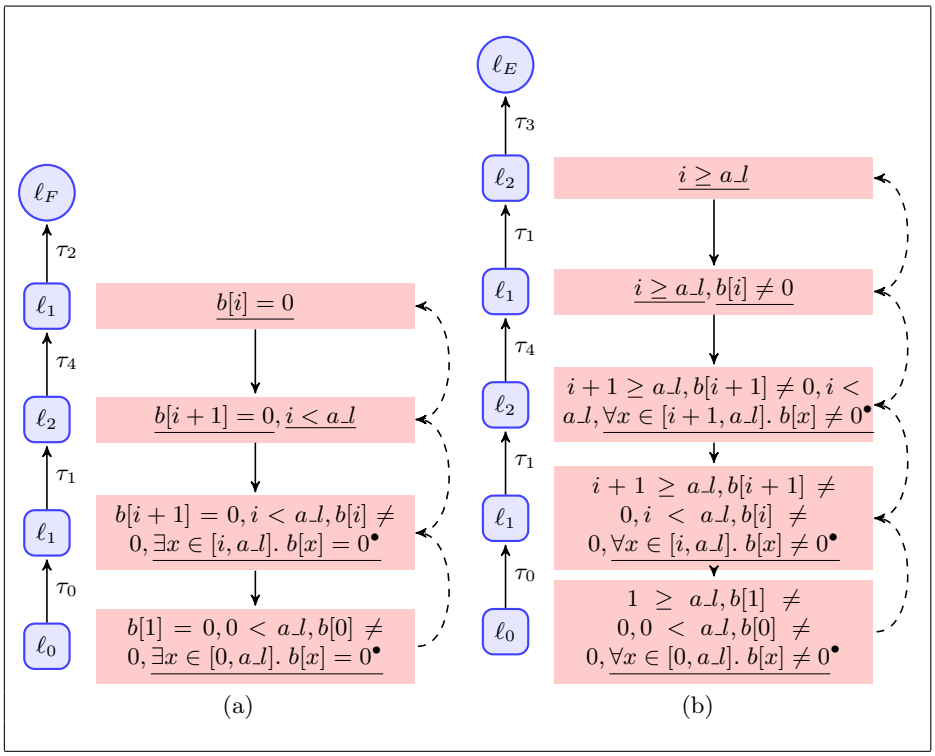

Fig. 3. Illustration of the refinement approach used in [29] on program copy. The underlined predicates are selected by the refinement process and predicates associate with the superscript $\bullet$ are computed using a system of inference rules.

one. With each state of the trace, is associated a set of predicates (in rectangular frames). Predicates without the $\bullet$ superscript, that we call *base* predicates, are obtained by computing the weakest precondition as shown by the solid arrows. Hence their conjunction represents the weakest precondition to reach the final location ℓ_F (respectively error location ℓ_E) in the safe trace (respectively error trace). The predicates associated with the superscript $\bullet$, called *general* predicates, are inferred using a generalisation procedure based on a system of inference rules as described in [29] (see Appendix A).

The details about the inference rules are not important to the contribution of this paper. The relevant point to retain is that we have a generalisation procedure able to infer new (general) predicates which logically represent consequences of the conjunctions of base predicates. For example, in the state belonging to the error trace (b) and associated with location ℓ_0, we have the general predicate $\forall x \in [0, a\_l].\ b[x] \neq 0$ which is a logical consequence of the base predicates at that state. We have $a\_l = 1$ due the predicates $1 \geq a\_l$ and $0 < a\_l$, and we have

$b[0] \neq 0$ and $b[1] \neq 0$, thus all elements of array b in the range $[1..a\_l]$ are not null.

The same procedure is applied to the safe trace (a). Once we reach the initial location, a minimisation procedure is called to keep only relevant predicates which are underlined. This procedure gives priority to general predicates. In our case, we keep $\exists x \in [0, a\_l].\ b[x] = 0$ and $\forall x \in [0, a\_l].\ b[x] \neq 0$ for the safe trace and unsafe one respectively at location ℓ_0 as they are the ones showing that the two traces are inconsistent.

The next step is to perform a dependency analysis starting from the two new states (retained predicates) and going forward in the direction of dashed arrows. Here also, we give priority to general predicates over base predicates. For example, in the state associated with location ℓ_1 of the safe trace, just before the initial state (location ℓ_0), we keep predicate $\exists x \in [i, a\_l].\ b[x] = 0$ as it is the one on which the predicate $\exists x \in [0, a\_l].\ b[x] = 0$ at location ℓ_0 depends.

A drawback of this approach is that it induces redundant computations. There are inter-trace redundancies due to shared parts between traces. For example, the two traces in Figure 3 are sharing a large part of their prefixes, namely $\langle \ell_0, \ell_1, \ell_2, \ell_1 \rangle$. There are also intra-trace redundancies due to backtracking, i.e, going backward for the predicate generation and forward for the dependency analysis. Our new approach remedies these weaknesses.

Eager-abstraction-based precondition inference. In our new approach the refinement process is completely skipped, states are explored backwards and predicates are added on the fly until a fix-point is reached. To be able to proceed so, we need first to find a node such that all the traces reaching it are common prefixes (going forward) for error traces and safe traces. Hence, such a node simply represents a common dominator for the error location and the final one. We choose the closest common dominator[4] as it maximises the length of common prefixes of traces. Up to that node, traces are explored separately, and from it and going further traces are explored conjointly.

This new scheme is illustrated in Figure 4. First, the two traces (safe and unsafe) are explored separately backwards up to the location ℓ_1 which represents a common dominator for the final location ℓ_F and the error location ℓ_E. At that point, we have $\varphi_F \equiv (b[i] = 0)$ as safe state, and $\varphi_E \equiv (i \geq a\_l \wedge b[i] \neq 0)$ as error state. We can see that φ_F and φ_E are inconsistent. Then, using the system of rules (from [29], see Appendix A), we try to infer a general predicate φ' from φ_F such that its negation $\neg\varphi'$ can be inferred from φ_E using the same system of rules. Hence we have $\varphi_F \Rightarrow \varphi'$ and $\varphi_E \Rightarrow \neg\varphi'$. If we find such a predicate φ', we keep it together with its negation $\neg\varphi'$ and throw all other predicates. It means that φ' becomes the new safe state and $\neg\varphi'$ the error one. Otherwise, we just keep all base predicates forming φ_F and φ_E.

Using the system of inference rules (Appendix A), we see that such a predicate (φ') cannot be inferred at the first encounter of location ℓ_1, so we keep all

[4] The closest common (or immediate) dominator for a set of nodes S is a node d which dominates S such that any other dominator d' for S is also a dominator for d.

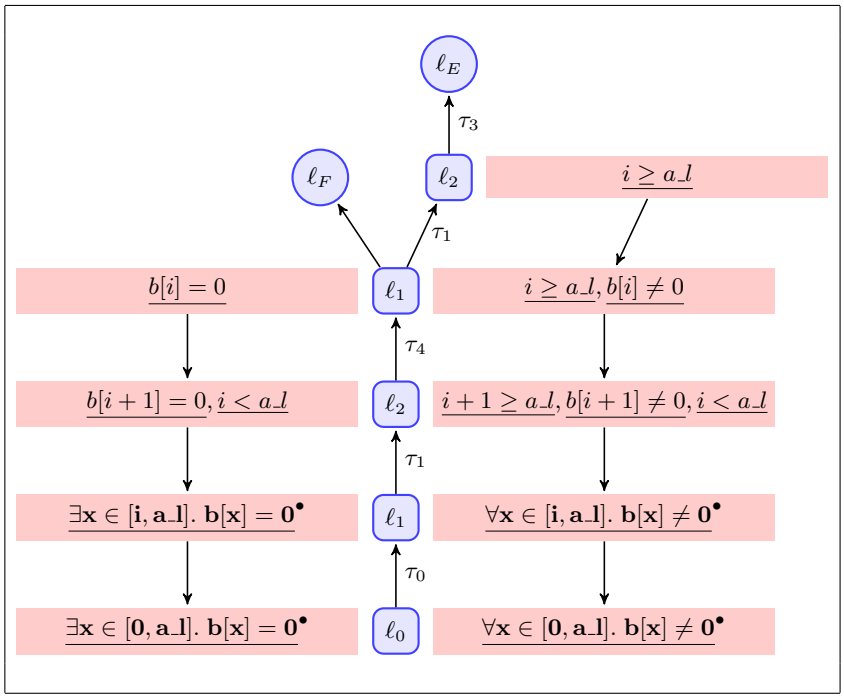

Fig. 4. Illustration of the new refinement approach based on analysing safe and unsafe traces conjointly. The underlined predicates are selected by the refinement process and predicates associate with the superscript $\bullet$ are computed using a system of inference rules.

base predicates. Going one step further using the weakest precondition, at location ℓ_2 we obtain $\varphi_F \equiv (b[i+1] = 0 \wedge i < a\_l)$ and $\varphi_E \equiv (i+1 \geq a\_l \wedge b[i+1] \neq 0 \wedge i < a\_l)$. From φ_E we infer $\forall x \in [i+1, a\_l].\ b[x] \neq 0$, but its negation $\exists x \in [i+1, a\_l].\ b[x] = 0$ cannot be inferred from φ_F via our system of inference rules. Again, we keep all base predicates and continue with the next step backwards. At the second encounter of location ℓ_1, this time we have $\varphi_F \equiv (b[i+1] = 0 \wedge i < a\_l \wedge b[i] \neq 0)$ and $\varphi_E \equiv (i+1 \geq a\_l \wedge b[i+1] \neq 0 \wedge i < a\_l \wedge b[i] \neq 0)$. From φ_E we can infer $\forall x \in [i, a\_l].\ b[x] \neq 0$ which represents φ' as its negation $\exists x \in [i, a\_l].\ b[x] = 0$ can be inferred from φ_F as well. Thus, we retain φ' and $\neg\varphi'$, and proceed further backwards with the same procedure as shown in Figure 4. We keep applying this procedure to generate states until reaching a fix-point (no new state found). At the end, we obtain the precondition $\exists x \in [0, a\_l].\ b[x] = 0$. Observe that this precondition is sufficient as having an element of array b in the interval $[0, a\_l]$ which is null guarantees the loop termination before violating the assertion condition. It is also necessary as its negation, $\forall x \in [0, a\_l].\ b[x] \neq 0$, allows the loop to iterate at least until the variable i becomes equal to a_l which causes the violation of the assertion.

We call this process *eager abstraction* as opposed to the lazy abstraction governing the CEGAR process. This procedure reduces intra-trace redundancies induced by the refinement which traverses traces back and forth to generate predicates and perform a dependency analysis. Here we can also decide at early stages about relevant predicates to keep, and states are monotonically generated. By construction, safe traces and error traces always contain enough information to show their inconsistency. The new procedure also reduces inter-trace redundancies as common prefixes are explored in parallel. All these points lead to a faster convergence of the fix-point computation.

In our experiments with the program in Fig. 1 including the side assertions to avoid null-pointer dereferencing of a and b (**assert**(a!=NULL); **assert**(b!=NULL) and access out of bounds for b (**assert**(i<b_l); in ℓ_1 and ℓ_2), we obtain the precondition $b \neq null \wedge (a \neq null \vee b[0] = 0) \wedge \exists x.\ 0 \leq x \leq a\_l \wedge x < b\_l \wedge b[x] = 0$ which is both necessary and sufficient for safe execution.

3 Preliminaries

In this section, we provide background on some ingredients used in our algorithm.

Program. A program is given as a set $\mathcal{TC}$ of transition constraints τ. A transition constraint τ is a formula of the form

$$g(X) \wedge \big(x'_1 = e_1(X)\big) \wedge \ldots \wedge \big(x'_n = e_n(X)\big) \tag{1}$$

where $X = \langle x_1, \ldots, x_n \rangle$ is a vector of program variables, which include the program counter *pc*. In (1), unprimed variables refer to the program state before performing the transition and primed ones represent the program state after performing the transition. Formula $g(X)$ is called the *guard* and the remaining conjuncts of τ are the *update* or *assignment*.

Representing states symbolically. Let us write $\mathcal{V} = \{x_1, \ldots, x_n\}$ for the set of variables of the program (including the program counter *pc*). For a variable $x \in \mathcal{V}$, $Type(x)$ is the type (range) of x and $\sigma(x)$ is a valuation of x such that $\sigma(x) \in Type(x)$. The variable *pc* ranges over the set of all program locations. For a vector X of variables, a program state is the valuation $\sigma(X) = \langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$.

A set of program states S is represented symbolically by means of the *characteristic function* of S. The formula φ represents the set of all those states that correspond to a satisfying assignment of φ, i.e., $\{\sigma(X) \,|\, \varphi[\sigma(X)/X]\}$[5]. We will use sets and their characteristic functions interchangeably. Symbolic states (formulas) are partially ordered via the implication operator $\Rightarrow$, i.e., $\varphi' \subseteq \varphi$ means $\varphi' \Rightarrow \varphi$.

[5] The notation $f[Y/X]$ represents the expression obtained by replacing all occurrences of every variable from the vector X in f with the corresponding variable (value) from Y. It naturally extends to a collection (set or list) of expressions.

State transformer. For a formula φ, the application of the operator pre with respect to the transition constraint τ returns a formula representing the set of all predecessor states of φ under the transition constraint τ, formally

$$\mathsf{pre}(\tau, \varphi(X)) =_{\mathsf{def}} g(X) \wedge \varphi[\langle e_1(X), \ldots, e_n(X)\rangle / X] \ .$$

For the whole program $\mathcal{TC}$, pre is given by

$$\mathsf{pre}(\varphi(X)) =_{\mathsf{def}} \bigvee_{\tau \in \mathcal{TC}} \mathsf{pre}(\tau, \varphi(X)) \ .$$

For a trace $\pi = \tau_1; \ldots; \tau_n$, we have

$$\mathsf{pre}(\tau_1; \ldots; \tau_n, \varphi) =_{\mathsf{def}} \mathsf{pre}(\tau_1, \ldots \mathsf{pre}(\tau_{n-1}, \mathsf{pre}(\tau_n, \varphi))) \ .$$

If $\mathsf{pre}(\pi, \varphi)$ is not equivalent to false, then the trace π is *feasible*.

(Un)Safe states. To ease the presentation, let us assume that the program contains a single error location ℓ_E and a single final location ℓ_F ($\ell_E \neq \ell_F$).[6] We denote by bad the set of *error states*, which is simply given by $pc = \ell_E$. Similarly, we call final the set of *final states*, which is represented by $pc = \ell_F$.

The set of safe states safe contains all states from which a final state is reachable. Formally,

$$\mathsf{safe} =_{\mathsf{def}} \mathsf{lfp}(\mathsf{pre}, \mathsf{final}) \tag{2}$$

where $\mathsf{lfp}(\mathsf{pre}, \varphi)$ denotes the least fix-point of the operator pre above φ. Similarly, unsafe is the set of all states from which an error (bad) state is reachable:

$$\mathsf{unsafe} =_{\mathsf{def}} \mathsf{lfp}(\mathsf{pre}, \mathsf{bad}) \ . \tag{3}$$

The least fix-points represent inductive backwards invariants, which we denote by ψ_{bad} and ψ_{final}, respectively. The invariants are inductive under pre, i.e.,

- $\mathsf{bad} \subseteq \psi_{\mathsf{bad}}$ and $\mathsf{final} \subseteq \psi_{\mathsf{final}}$
- $\mathsf{pre}(\psi_{\mathsf{bad}}) \subseteq \psi_{\mathsf{bad}}$ and $\mathsf{pre}(\psi_{\mathsf{final}}) \subseteq \psi_{\mathsf{final}}$

In the absence of non-determinism in the program, the sets of unsafe and safe states are disjoint, and we have

$$\mathsf{unsafe} \wedge \mathsf{safe} = \mathsf{false} \ .$$

Predicate abstraction. Predicate abstraction consists in approximating a state φ with a formula φ' constructed as a Boolean combination of predicates taken from a set P. Here, the term approximation means that any model that satisfies φ must satisfy φ'. Thus, a suitable approximation is obtained via the logical

[6] In case of multiple assertions, we add an edge from each assertion (guarded with the negation of the assertion) to ℓ_E. Similar treatment can be applied in the case of multiple return locations.

implication "$\Rightarrow$", i.e., φ' is the strongest Boolean combination built up from predicates taken from the finite set P such that $\varphi \Rightarrow \varphi'$.

Defining the abstraction function α as being the strongest Boolean combination of predicates in P is not practical because of the exponential complexity of the problem. Therefore, we use a lightweight version of α that builds the strongest conjunction of predicates in P:

$$\alpha(\varphi) =_{\mathsf{def}} \bigwedge p \quad s.t. \quad p \in P \wedge \varphi \Rightarrow p \; .$$

Let us have $\mathcal{D}^\sharp$ the domain of formulas built up from the finite set of predicates P. The domain $\mathcal{D}^\sharp$ is not closed under pre, therefore, we define $\mathsf{pre}^\sharp$ under which $\mathcal{D}^\sharp$ is closed. Let us associate the concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ to α, we simply choose γ to be the identity function. Functions α and γ form a *Galois connection* with respect to $\subseteq$ ($\Rightarrow$) being the partial order relation for both $\mathcal{D}$ and $\mathcal{D}^\sharp$. Formally speaking

$$\forall x \in \mathcal{D} \; \forall y \in \mathcal{D}^\sharp . \; \alpha(x) \subseteq y \Leftrightarrow x \subseteq \gamma(y) \; .$$

Hence, we define $\mathsf{pre}^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, the abstract version of pre, as follows:

$$\mathsf{pre}^\sharp(\varphi) =_{\mathsf{def}} \alpha(\mathsf{pre}(\gamma(\varphi))) \; ,$$

and thus

$$\mathsf{pre}^\sharp(\tau, \varphi) = \alpha(\mathsf{pre}(\tau, \varphi)) = \bigwedge p \quad s.t. \quad p \in P \wedge \mathsf{pre}(\tau, \varphi) \Rightarrow p \; .$$

Moreover, for a disjunction we have

$$\mathsf{pre}^\sharp(\tau, \bigvee_{j \in J} \varphi_j) =_{\mathsf{def}} \bigvee_{j \in J} \mathsf{pre}^\sharp(\tau, \varphi_j) \; .$$

As seen for pre, the operator $\mathsf{pre}^\sharp$ also extends to traces. Henceforth, whenever we write $\mathsf{pre}^\sharp_P$ we mean that the abstraction (image) is computed by considering predicates from the set P.

The lattice of abstract states $(\mathcal{L}, \Rightarrow)$ is finite as the set of predicates is finite. Therefore, $\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{bad})$ (resp. $\mathsf{lfp}(\mathsf{pre}^\sharp, \mathsf{final})$), the least fixpoint for $\mathsf{pre}^\sharp$ above bad (resp. final) in $\mathcal{L}$, is computable.

4 Eager abstraction

In this section, we present our approach for the inference of necessary and sufficient preconditions. We recall that a necessary and sufficient precondition φ is a precondition under which no error trace is feasible, and no safe trace is excluded. In other words, it is neither too strong nor too weak. As mentioned previously (Section 3), it is not always possible to compute the set of safe (or unsafe) states using the weakest precondition transformer pre. Therefore, we use its abstract

Algorithm 1: EagerPrecond

Input: set of transition constraints (program) $\mathcal{TC}$
Output: formula (precondition)

Var P: set of predicates;
Var ψ_F, ψ_E: formulas;
Find a common dominator node ℓ_d for locations ℓ_F and ℓ_E;
Let φ_{dF} be the necessary and sufficient precondition for final at ℓ_d;
Let φ_{dE} be the necessary and sufficient precondition for bad at ℓ_d;
if $\varphi_{dF} \wedge \varphi_{dE} \not\equiv \mathsf{false}$ **then abort** "failure";
;
$\psi_F := \varphi_{dF}$;
$\psi_E := \varphi_{dE}$;
while true **do**
  $\psi_F^0 := \psi_F$;
  $\psi_E^0 := \psi_E$;
  foreach $\tau \in \mathcal{TC}$ **do**
    $P := \mathsf{SplitPreds}(\tau, \psi_F, \psi_E)$;
    if $P = \emptyset$ **then abort** "failure";
    ;
    $\psi_F := \psi_F \vee \mathsf{pre}_P^\sharp(\tau, \psi_F)$;
    $\psi_E := \psi_E \vee \mathsf{pre}_P^\sharp(\tau, \psi_E)$;
  if $\psi_F \subseteq \psi_F^0 \wedge \psi_E \subseteq \psi_E^0$ **then return** ψ_F;
  ;

version $\mathsf{pre}^\sharp$. Formally speaking, our goal is to find a set of predicates P which allows us to compute φ such that the two constraints below are fulfilled:

$$\mathsf{lfp}(\mathsf{pre}_P^\sharp, \mathsf{final}) \subseteq \varphi \text{ (no exclusion of safe states)} \tag{4}$$

$$\mathsf{lfp}(\mathsf{pre}_P^\sharp, \mathsf{bad}) \wedge \varphi \equiv \mathsf{false} \text{ (no inclusion of unsafe states)} \tag{5}$$

As opposed to our previous work [29], our goal here is to compute φ monotonically in a single pass. To this end, our algorithm needs to have some features such as:

- A guidance criterion so that at each state exploration step, the two constraints (4) and (5) hold as an invariant of our algorithm.
- The previous point implies inferring predicates on the fly, as fixing predicates in advance reduces the ability of building abstractions satisfying (4) and (5) at each step.

Hence, we explore safe and unsafe states in parallel taking into account their disjointness as condition that must hold at each step. This idea is translated to the Algorithm $\mathsf{EagerPrecond}$ (Algorithm 1).

In the algorithm, the set of safe and unsafe states are symbolically represented via the formulae ψ_F and ψ_E. As the final location ℓ_F and error location ℓ_E are

separate, the first question to be answered is: from which location do we start exploring safe and unsafe states conjointly? We choose this location to be the dominator location ℓ_d which is common to ℓ_F and ℓ_E, as any program trace must go through it to reach any of them.

Remark 1. Computing the necessary and sufficient precondition for reaching ℓ_F and ℓ_E (lines 4 and 5 of Algorithm 1) from the dominator node ℓ_d is in practice straightforward, as in most of the programs we have tested, all the paths leading from the dominator location to ℓ_F and ℓ_E are loop-free. However, in the presence of loops, we can use our CEGAR-based technique [29] to compute the precondition up to ℓ_d and then apply the eager approach.

We then compute the weakest precondition to reach ℓ_F and ℓ_E (lines 4 and 5 of Algorithm 1) which respectively gives φ_{dF} and φ_{dE} and they should be disjoint (see line 6). From location ℓ_d onward, states are explored conjointly by taking each time the same transition τ (lines 15 and 16). This step depends on the set of predicates computed by calling the procedure SplitPred at line 13. The set of predicates P is computed in a way that the new abstractions obtained via $\mathsf{pre}^\sharp_P$ are disjoint. This process is iterated until no new safe or unsafe states are discovered (line 17).

Remark 2. The formula ψ_F computed by Algorithm 1 represents all the states which potentially reach the final location from different program locations. To get the precondition at the initial location ℓ_0, it suffices to project ψ_F on ℓ_0, which is simply expressed by the formula $\psi_F \wedge pc = \ell_0$.

SplitPred. Let us now have a look inside the procedure SplitPred (Algorithm 2). The role of this procedure is to deliver the set of predicates under which the next computed abstractions fulfil the separation criterion between safe and unsafe states. It takes as parameters two formulae ψ_F and ψ_E and a transition constraint τ, and returns a set of predicates P such that

$$(\psi_F \vee \mathsf{pre}^\sharp_P(\tau, \psi_F)) \wedge (\psi_E \vee \mathsf{pre}^\sharp_P(\tau, \psi_E)) \equiv \mathsf{false} \tag{6}$$

In other words, the over-approximations of the predecessor sets with respect to the transition constraint τ and the set of predicates P are disjoint. First, the exact predecessor sets are computed using the pre operator (lines 3 and 4 in Algorithm 2), if the resulting formulae are not disjoint, there is no need to go further (line 5) as the abstraction will make them even weaker.

We are then interested in the states associated with the program location given by $\ell = \tau.pc$[7] as they are the potentially newly generated ones obtained via transition τ. We obtain this subset by projecting each global set of states on the location ℓ as shown at lines 7 and 8. These sets $\psi_{F\ell}$ and $\psi_{E\ell}$ are disjunctions of formulae, such that every disjunct (φ_i's and φ'_j's) represents a symbolic state

[7] The notation $\tau.pc$ simply refers to the program counter value in the pre-state of the transition τ.

Algorithm 2: SplitPred

Input: formula ψ_F, ψ_E, transition constraint τ
Output: set of predicates

```
1  Var P, P': set of predicates;
2  Var ψ_F, ψ_E: formula;
3  ψ_F := ψ_F ∨ pre(τ, ψ_F);
4  ψ_E := ψ_E ∨ pre(τ, ψ_E);
5  if ψ_F ∧ ψ_E ≢ false then return ∅;
6  ;
7  Let τ.pc = ℓ;
8  Let ψ_Fℓ ≡ (ψ_F ∧ pc = ℓ) be of the form ⋁_(i∈I) φ_i;
9  Let ψ_Eℓ ≡ (ψ_E ∧ pc = ℓ) be of the form ⋁_(j∈J) φ'_j;
10 P := ∅;
11 foreach (i, j) ∈ I × J do
12     if ∃ p s.t (p ∈ InferGen(φ_i) ∧ ¬p ∈ InferGen(φ'_j)) then P := P ∪ {p, ¬p};
13     ;
14     else P := P ∪ atoms(φ_i) ∪ atoms(φ'_j);
15     ;
16 return P;
```

and is a conjunction of predicates according to our definition of the predicate transformer (see Section 3).

For each pair of states (φ_i, φ'_j) respectively belonging to the set of safe states and unsafe ones at location ℓ, we try to extract general predicates which they induce using the procedure InferGen (line 11). The extraction of new predicates is based on the system of inference rules [29] (see Appendix A).

If there exists a general predicate p which can be inferred from φ_i and its negation $\neg p$ can be inferred from φ'_j, then it is selected together with its negation (line 11). In fact, p is implied by φ_i and is inconsistent with φ'_j (i.e., $\varphi'_j \wedge p \equiv false$) as $\neg\varphi$ is implied by φ'_j. Hence both p and $\neg p$ are good potential candidates for building new separate states. If we cannot infer such a predicate p, then we return the set of atoms forming the two states (line 12), which keeps the resulting states separated. The function atoms is simply defined as $\mathsf{atoms}(\varphi_1 \wedge \ldots \wedge \varphi_n) = \{\varphi_1, \ldots, \varphi_n\}$. It takes a conjunction as argument and returns the set of conjuncts forming it.

Remark 3. Note that SplitPred returns an interpolant [20] for the predecessors of the two formulae taken as parameters. Hence, we could also use an interpolation procedure as a replacement of SplitPred. The investigation of this possibility is left for future work.

Proposition 1. *The formula computed by Algorithm 1 is a necessary and sufficient precondition, i.e. it satisfies (4) and (5).*

Proof. Let us denote by φ the formula returned by Algorithm 1. For (4), φ (which represents ψ_F) is a fix-point according to the termination criterion at line 17 of

Algorithm 1. For (5), we have $\mathsf{lfp}(\mathsf{pre}^\sharp_P, \mathsf{bad}) \subseteq \psi_E$. Also ψ_E is inconsistent with ψ_F as they are initially inconsistent (line 6 of Algorithm 1) and all updates at lines 15 and 16 based on the set of predicates returned by SplitPred satisfies (6), hence ψ_E and ψ_F remain inconsistent. Thus $\mathsf{lfp}(\mathsf{pre}^\sharp_P, \mathsf{bad}) \wedge \varphi \equiv \mathsf{false}$.

Discussion. Our algorithm aborts if it fails to infer a necessary and sufficient precondition (see lines 6 and 14 in Algorithm 1). This can happen due to several reasons: (1) If the program is non-deterministic then $\psi_F \wedge \psi_E$ might be satisfiable; (2) if the predicates inferred by the inference rules do not give rise to sufficiently precise loop invariants to guarantee separation of ψ_F and ψ_E; or (3) if the SMT solver that we are using is unable to conclude unsatisfiability of $\psi_F \wedge \psi_E$. We chose the SMT solver Z3 for our experiments because it did not exhibit any problems regarding (3). However, we encountered some issues regarding the handling of quantifiers in preliminary experiments with other SMT solvers.

Our algorithm cannot distinguish between terminating and non-terminating traces. The problem to perform such a distinction is known as *conditional termination*, i.e. computing preconditions that ensure termination. The extension of our algorithm in this respect is a direction of future work that we pursue.

A well-known problem in predicate abstraction-based methods is non-termination of the analysis if the predicate generalisation method fails to generate the required loop invariants. Common approaches to force termination are the introduction of aggressive generalisation rules (like widening in abstract interpretation) that guarantee that our algorithm eventually answers "failure", or the restriction to a finite predicate language [23] (corresponding to a finite height domain in abstract interpretation). However, the latter method spoils the advantage of our approach that the predicate language adapts itself to the program being analysed, and due the reduced expressiveness our algorithm would answer "failure" more often.

5 Experimental results

We have implemented our precondition inference technique in the P-Gen[8] tool which takes as input a C program containing a procedure annotated with an assertion to be verified and returns a necessary and sufficient precondition for the validity of the specified assertion.

We performed experiments using a desktop computer with 3.7 GB of RAM and a Core 2 processor with 3 GHz, running Linux. P-Gen uses several theorem provers, such as Yices [17], Simplify [16] and Z3 [15], to compute the abstraction and analyse counterexamples. We used Z3 in our experiments as we noticed that it is the one which handles quantifiers best compared to the other theorem provers.

The results of our experiments are illustrated in Table 1. The column "Precondition" shows the type of precondition inferred, "Q" stands for quantified

[8] http://www.cs.ox.ac.uk/people/nassim.seghir/pgen-web-page

Program	Precondition	Predicates		Time (s)	
		Eager	CEGAR	Eager	CEGAR
strncmp	Q + S	12	20	**17.54**	**536.70**
strcat	Q + S	3	4	0.18	0.55
memchr	Q + S	7	4	**4.28**	**64.42**
strlen	Q + S	3	4	0.18	0.54
memcpy	S	4	3	0.063	0.15
strchr	Q + S	6	8	0.65	1.76
r_strcat	Q	5	2	1.08	8.04
r_strncpy	Q + S	7	4	**2.90**	**253.55**
strcspn	Q + S	4	3	0.30	0.57
strspn	Q + S	4	3	0.31	0.56
my_strcmp	Q + S	6	7	0.66	2.35
my_memcmp	Q + S	7	5	3.46	20.9
AllNotNull	Q + S	4	3	0.30	2.07
mvswap	S	3	2	0.056	0.061

Table 1. Experimental comparison between eager abstraction and the counterexample-guided approach (from [29]).

and "S" stands for simple (quantifier-free). The column "Predicates" represents the number of predicates inferred to abstract the set of unsafe (safe) states. As we are associating different sets of predicates with different locations, similar to [21], we provide the average number of predicates per location instead of the total number of predicates. Both columns "Time" and "Predicates" are divided into two columns "CEGAR" which represents the counterexample-guided precondition inference approach [29], and "Eager" which refers to the current approach. These examples are implementations of routines from the C string library[9]. The assertions ensure freedom of runtime errors like null pointer dereferencing and array-out-of-bounds accesses. All the benchmarks used as well as the (runs) results of the comparative study are available online[10]. The generated preconditions for the different examples are included as well.

The preconditions generated by the two approaches are semantically equivalent for all these benchmarks, but syntactically different in some cases (due to redundancies).

We can see that the eager approach clearly outperforms the CEGAR-based one in all the cases. This difference is even clearer for programs strncmp, memchr and r_strncpy as the running time takes minutes for the CEGAR approach while it does not go beyond 18 seconds for the eager one. This is encouraging and demonstrates the relevance and practicality of our new approach.

[9] An implementation of the different functions is available here: http://en.wikibooks.org/wiki/C_Programming/Strings

[10] http://homepages.inf.ed.ac.uk/mseghir/benchmarks_and_results_aplas14.tar.gz

6 Related Work

The combination of predicate abstraction [19] with counterexample-guided abstraction refinement [9] has been implemented in many tools [2, 8, 10, 21, 22, 26]. Most of them use CEGAR to check the validity of a given assertion. We go beyond that by finding the precondition under which the assertion is valid.

Some other tools are inspired by Hoare's reasoning style [3, 14, 18]. They are based on the reasoning-by-contract principle: pre- and postconditions and loop invariants have to be specified by the user, which is a tedious task in general. Our technique can support the user by generating preconditions for less interesting side verification obligations (internal assertions), allowing him to focus on the functional aspect (postcondition) of the verification task.

Moy [25] proposed a technique to infer preconditions. While his technique is stronger than many existing ones, it is unable to infer quantified preconditions. Our technique infers universally as well as existentially quantified preconditions for array programs.

Blanc and Kroening [4] proposed an approach for precondition generation to optimise the simulation of SystemC code. However, they have no guarantee that the inferred precondition is necessary and sufficient. Taghdiri [30] proposed an approach for generating approximations of relations (over pre- and post-states) induced by functions by bounding the number of loop unrolling, making the approach unsuitable for proving the absence of bugs. Our technique over-approximates the set of all (even infinite) behaviours. Thus, a computed precondition in our case guarantees safety.

Sankaranarayanan et al. [28] presented a technique that combines test and machine learning to infer likely data preconditions. The results obtained by their approach are promising. However, their technique can only suggest preconditions but does not guarantee their validity.

In the context of abstract interpretation, Cousot et al. [13] formulated precisely the contract inference problem for intermittent assertions. The preconditions extracted by their method are necessary preconditions, i.e. they do not exclude unsafe runs. In a later work [12], they took into account the calling context to identify under which circumstances a generated necessary precondition is also sufficient. We compute necessary and sufficient preconditions independently from the calling context of the procedure. Similar techniques for computing necessary preconditions are proposed by Miné [24] using a *lower widening* technique to perform a polyhedral backward analysis, and Bakhirkin et al [1] who combine over-approximative backward analysis with a *subtraction* operation to obtain under-approximations.

The method described in [5, 13, 24, 27] rely on predefined abstract domains. Thus, if the domain is not precise enough, either it is redesigned or another domain is used. In our approach, predicates are inferred syntactically on the fly and the only a priori restriction are the inference rules that are applied to generalise predicates to potential loop invariants. The inference mechanism can be enhanced by introducing new inference rules without having to implement new abstract program transformers. The advantage of this approach is that the

domain adapts itself to the program analysed. However, as discussed above, there is no guarantee for termination if the inference rules fail to generate the required loop invariants.

Calcagno et al. [7] presented a technique based on *bi-abduction* to infer pre- and post-specifications of heap structures. Although we can deal with pointers, the properties handled by their technique are out of the scope for our tool as we do not have a theory to reason about heap properties. On the other hand, the preconditions they compute are only necessary, hence false alarms are not ruled out.

Our current approach deals with the precondition generation problem in the context of safety. Extending it to the liveness context such as termination [6,11] is an area we are interested in for future work.

7 Conclusion

In this paper, we have presented an eager abstraction technique for generating necessary and sufficient preconditions. The idea underlying eager abstraction is that the invariant of separating safe and unsafe states is satisfied throughout the algorithm. Hence the abstraction process is monotone and no refinement is required.

The comparative study with our CEGAR-based approach for precondition generation demonstrates that our new method is a significant improvement and represents the right way to proceed for practicability. For all the programs we have tested, the eager approach performs better than the lazy (CEGAR) one. For cases where the lazy approach takes several minutes, the eager one just requires several seconds (< 18s). This is essential since precondition generation is mostly used in interactive development and verification environments, where response time is crucial for the practicability, productivity and the adoption of the environment by verification engineers.

References

1. A. Bakhirkin, J. Berdine, and N. Piterman. Backward analysis via over-approximate abstraction and under-approximate subtraction. In *Static Analysis Symposium*, volume 8723 of *Lecture Notes in Computer Science*. Springer, 2014.
2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
4. N. Blanc and D. Kroening. Race analysis for systemc using model checking. *ACM Trans. Design Autom. Electr. Syst.*, 15(3), 2010.
5. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*, pages 46–55, 1993.
6. M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2012.

7. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
8. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
10. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *TACAS*, pages 570–574, 2005.
11. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2008.
12. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.
13. P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, pages 150–168, 2011.
14. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE Companion*, pages 429–430, 2009.
15. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
16. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Lab., 2003.
17. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94, 2006.
18. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
19. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
22. F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-soft. In *ICCD*, pages 297–308, 2005.
23. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
24. A. Miné. Inferring sufficient conditions with backward polyhedral under-approximations. *ENTCS*, 287:89–100, 2012.
25. Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, pages 188–202, 2008.
26. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, volume 4354 of *LNCS*, pages 245–259, 2007.
27. X. Rival. Understanding the origin of alarms in astrée. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005.
28. S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, pages 295–306, 2008.

29. M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 451–471. Springer, 2013.
30. M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.

A Inference Rules

The system of rules in Figure 5 was proposed in [29] to generalise predicates. Among the symbols used in the system, e refers to linear terms, x is a variable and φ is a formula.

The rule ELIM linearly combines two constraints to eliminate common variables. Rule EQ infers equality constraints, which might be used by rule SUB to substitute occurrences of variables with equal terms. The rule UNIV builds a quantified formula and LINK bridges the intervals of two quantified formulas. Finally, the rule EXIST produces two existentially quantified formulas and the rules EXT_R and EXT_L extend the interval of an existentially quantified formula from the right and the left, respectively.

$$\frac{c_1.e + e_1 \geq 0 \ , \ -c_2.e + e_2 \geq 0}{\begin{array}{c} c_2.e_1 + c_1.e_2 \geq 0 \\ (c_1, c_2 > 0) \end{array}} \ (\text{ELIM}) \qquad \frac{x - e \geq 0 \ , \ -x + e \geq 0}{x = e} \ (\text{EQ})$$

$$\frac{\varphi(x) \ , \ x = e}{\varphi(e)} \ (\text{SUB}) \qquad \frac{\varphi(i), \ \neg\varphi(j) \ (i < j)}{\exists x \in \{i, \ldots, j\}.\ \varphi(x), \ \exists x \in \{i, \ldots, j\}.\ \neg\varphi(x)} \ (\text{EXIST})$$

$$\frac{\exists x \in \{i, \ldots, j\}.\ \varphi(x), \ j \leq k}{\exists x \in \{i, \ldots, k\}.\ \varphi(x)} \ (\text{EXT\_R}) \qquad \frac{\exists x \in \{i, \ldots, j\}.\ \varphi(x), \ k \leq i}{\exists x \in \{k, \ldots, j\}.\ \varphi(x)} \ (\text{EXT\_L})$$

$$\frac{\varphi(i)}{\forall x \in \{i\}.\ \varphi(x)} \ (\text{UNIV}) \qquad \frac{\forall x \in \{j, \ldots, i\}.\ \varphi(x) \ , \ \forall x \in \{i+1, \ldots, k\}.\ \varphi(x)}{\forall x \in \{j, \ldots, k\}.\ \varphi(x)} \ (\text{LINK})$$

i and j are integer variables appearing in a linear index expression in φ ($\neg\varphi$).

Fig. 5. Rules for general predicate inference